King Mongkut's University of Technology Thonburi

Sorting Problem

CSC 209 Data Structures

Dr. Kulwadee Somboonviwat

School of Information Technology, KMUTT

kulwadee.som [at] sit.kmutt.ac.th

Lecture Plan

- Sorting Problem
- Insertion Sort
- Divide-and-Conquer
- Mergesort
- Quicksort
- Implementation of insertion sort, mergesort, quicksort

Sorting Problem

Input: A sequence of n numbers $<a_1, a_2, ..., a_n>$

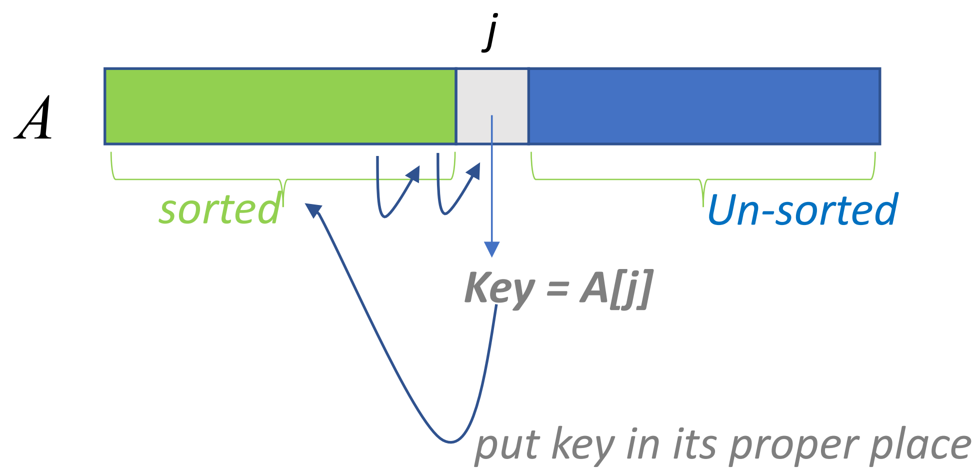
Output: A permutation (reordering) $<a'_1, a'_2, ..., a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

Example:

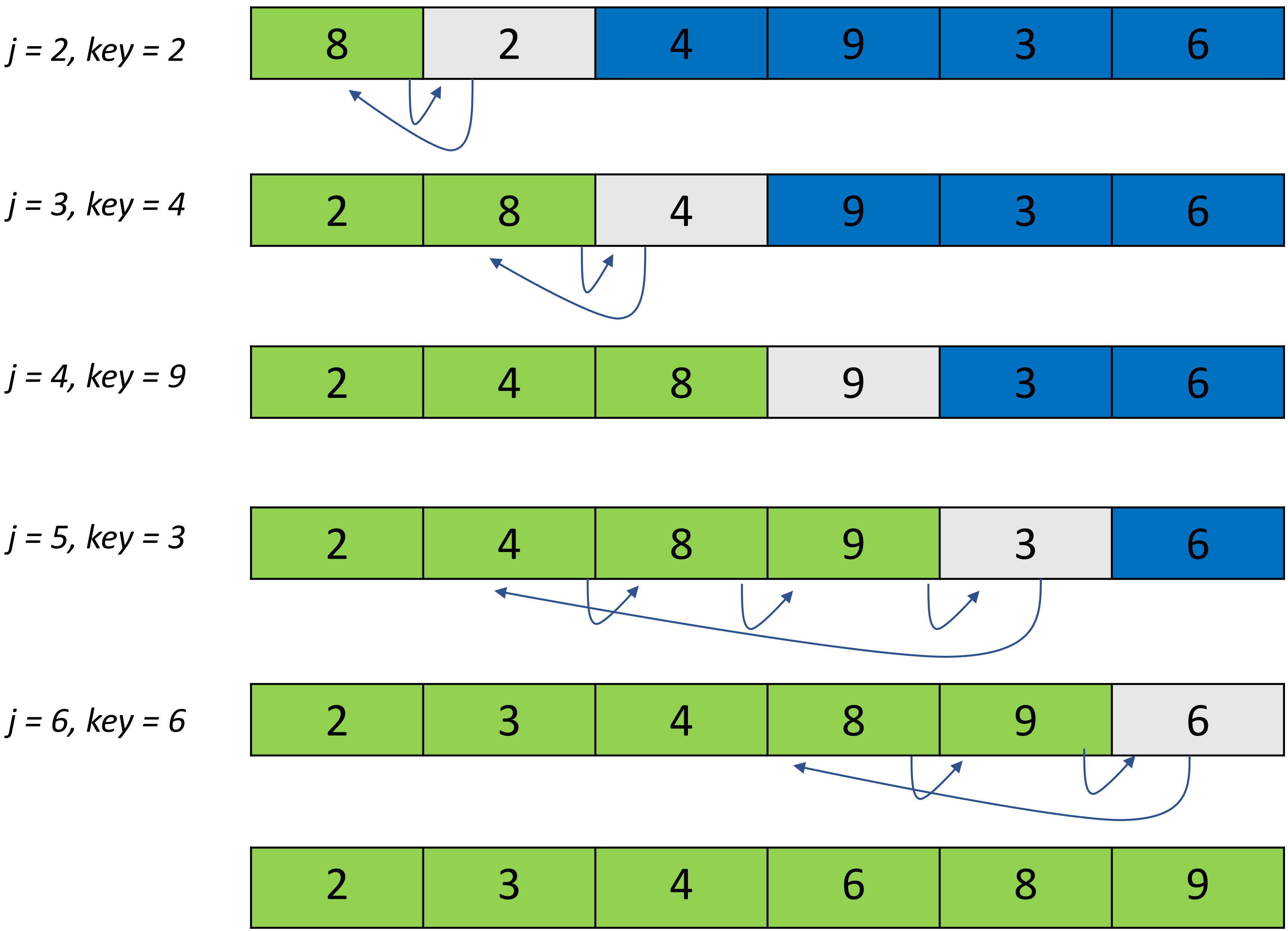
Input	8	2	4	9	3	6
Output	2	3	4	6	8	9

Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```



Insertion Sort



Running Time: Insertion Sort

- Depends on the input (e.g. sorted, reverse sorted)
- Depends on the input size
- In general, we want to know the upper bound of the running time because it represents a guarantee

Types of Running Time

- Worst-case running time

  T(n) : the maximum time on any input of size n

- Average-case running time

  T(n) : the expected time over all inputs of size n

  ** need an assumption of statistical distribution of inputs (e.g. the uniform distribution)*

- Best-case running time

  T(n) : the minimum time on any input of size n

  ** we are not so interested in the best-case because a slow algorithm might have a very good best-case running time*

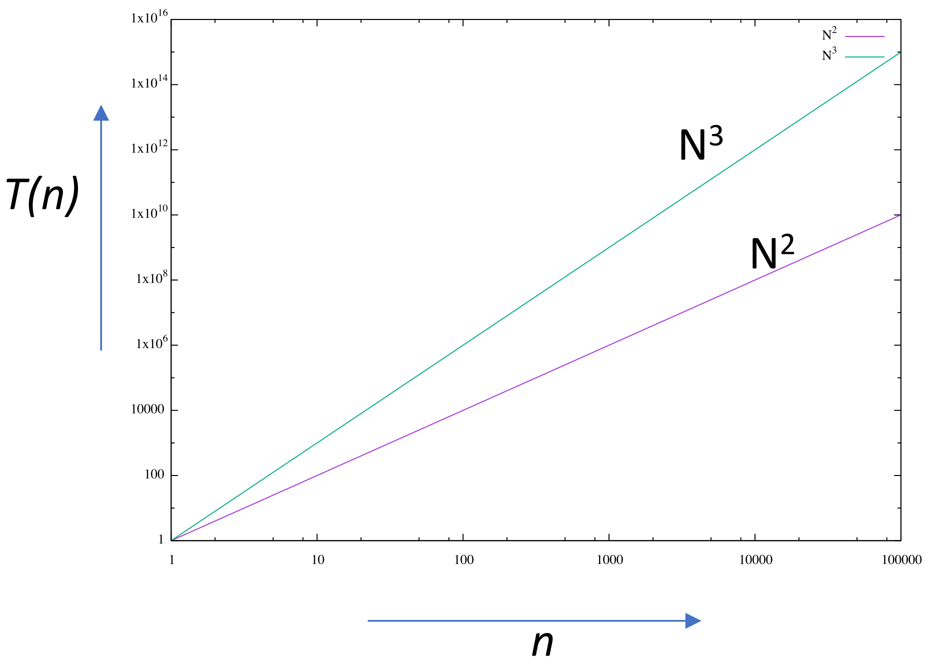
What is Insertion Sort's Running Time?

- Running time of an algorithm depends on many factors: computer hardware, compiler, operating systems, …

- BIG IDEA: Asymptotic Analysis
  - Ignore machine dependent constants
  - Look at the *rate of the growth of $T(n)$ as $n \rightarrow \infty$*

Asymptotic Analysis : Theta-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$ [1]



$$(3n^3 + 2n + 2) = \theta(n^3)$$

$$(4n^2 + 3n + 4) = \theta(n^2)$$

When n gets large enough, $\theta(n^2)$ algorithm ***always*** *beat $\theta(n^3)$ algorithm*

What is Insertion Sort's Running Time?

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$\quad i = j-1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\quad\quad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\quad\quad i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .
\end{aligned}
$$

- Worst-case running time

  (reverse sorted input):

  T(n) = $\sum_{j=2}^{N} \theta(j) = \theta(N^2)$

- Average-case running time

  (all ordering is equally likely):

  T(n) = $\sum_{j=2}^{N} \theta(j/2) = \theta(N^2)$

- Best-case running time

  (sorted input):

  T(n) = $\sum_{j=2}^{N} \theta(j) = \theta(N)$

Is Insertion-sort a fast algorithm?

- Moderately fast for small input size *n*
- Not at all for large input size *n*

Pop-Quiz

- Illustrate the operation of insertion-sort on the array

  $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$

Summary: Insertion Sort

- Worst case running time is $\theta(N^2)$
- An excellent sorting method for partially sorted arrays
- Moderately fast for small arrays
- Sort “in place”
- Very slow for large arrays

Divide-and-Conquer

- **Divide**: the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer**: the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
- **Combine**: the solutions to the sub-problems into the solution for the original problem.

Mergesort

- **Divide**: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- **Conquer**: Sort the two subsequences recursively using merge sort.
- **Combine**: ***Merge*** the two sorted subsequences to produce the sorted answer.

Mergesort

8	2	4	9	3	6	5	1

Mergesort

8	2	4	9	3	6	5	1

8	2	4	9

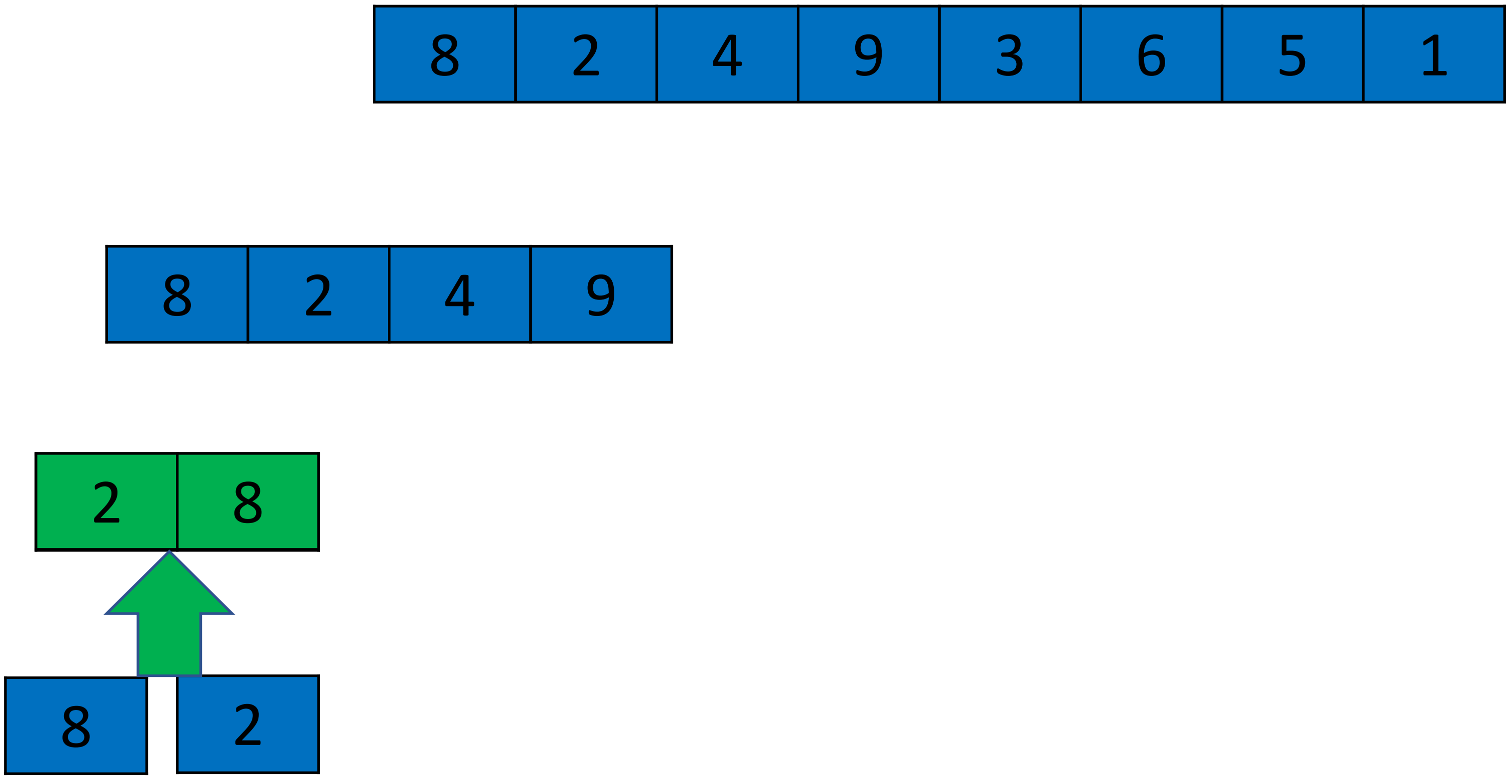
Mergesort

8	2	4	9	3	6	5	1

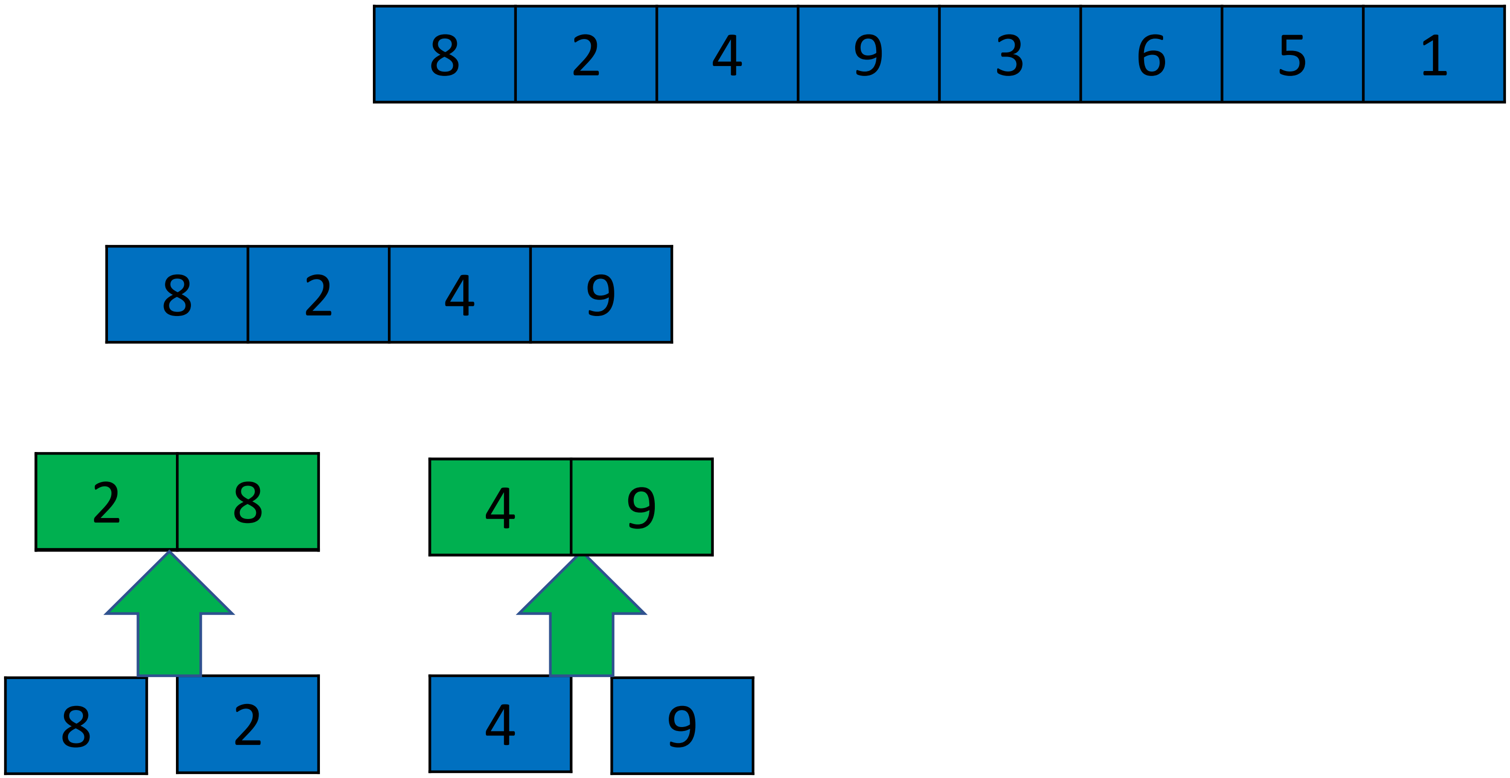
8	2	4	9

8	2

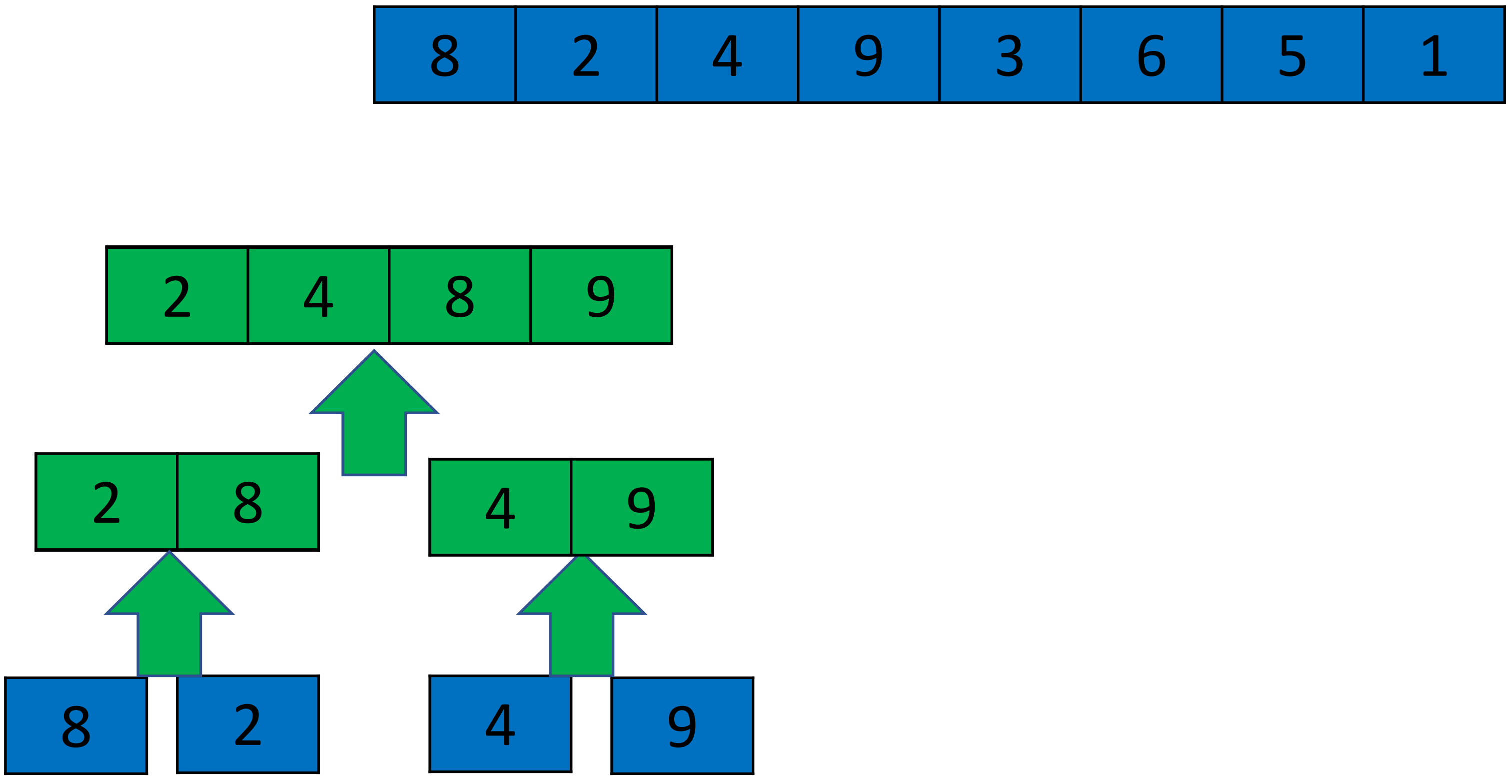
Mergesort



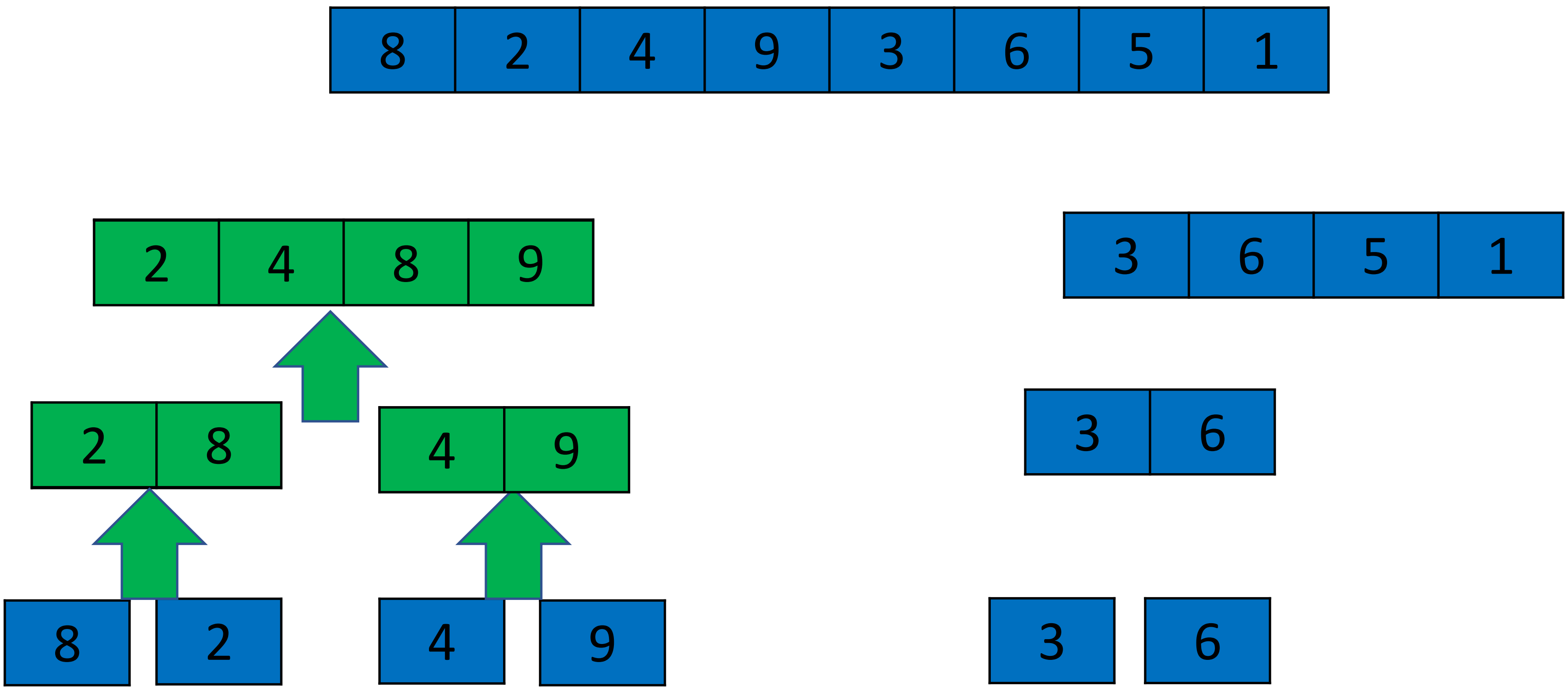
Mergesort



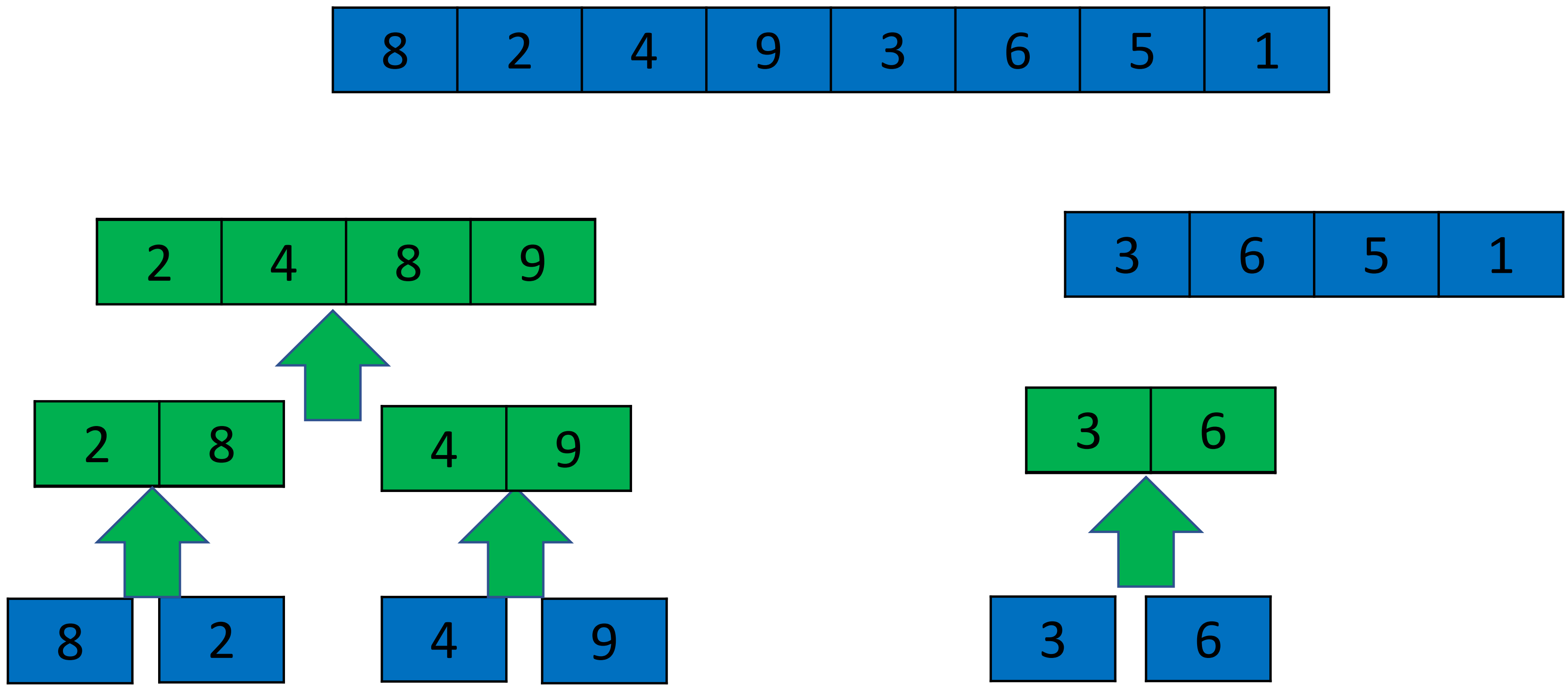
Mergesort



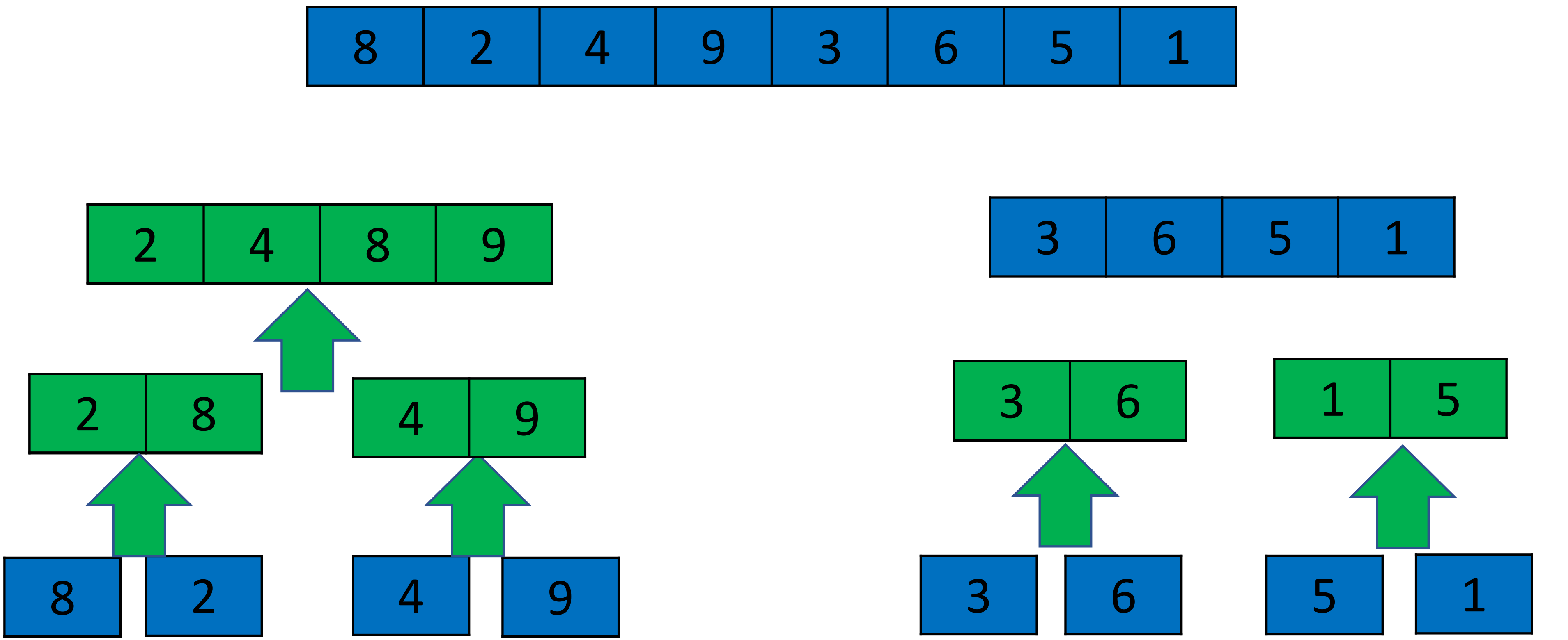
Mergesort



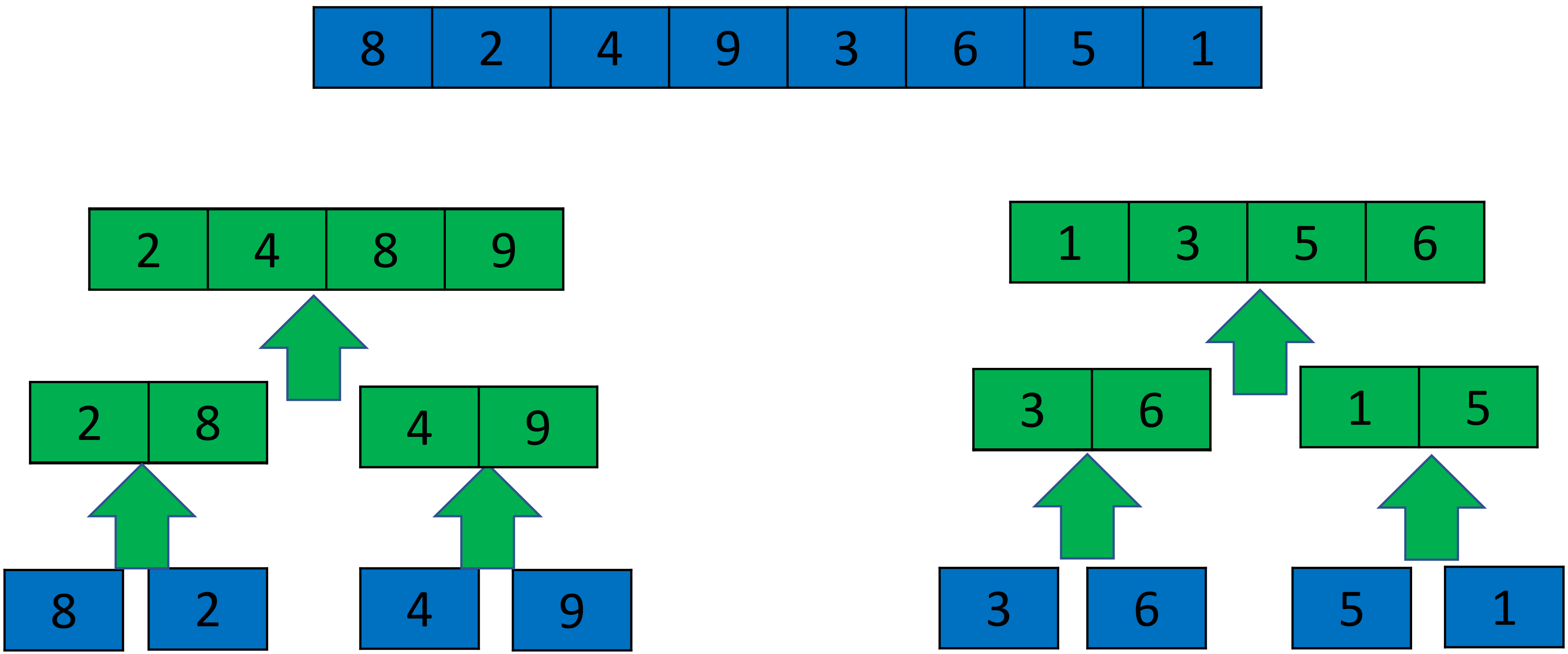
Mergesort



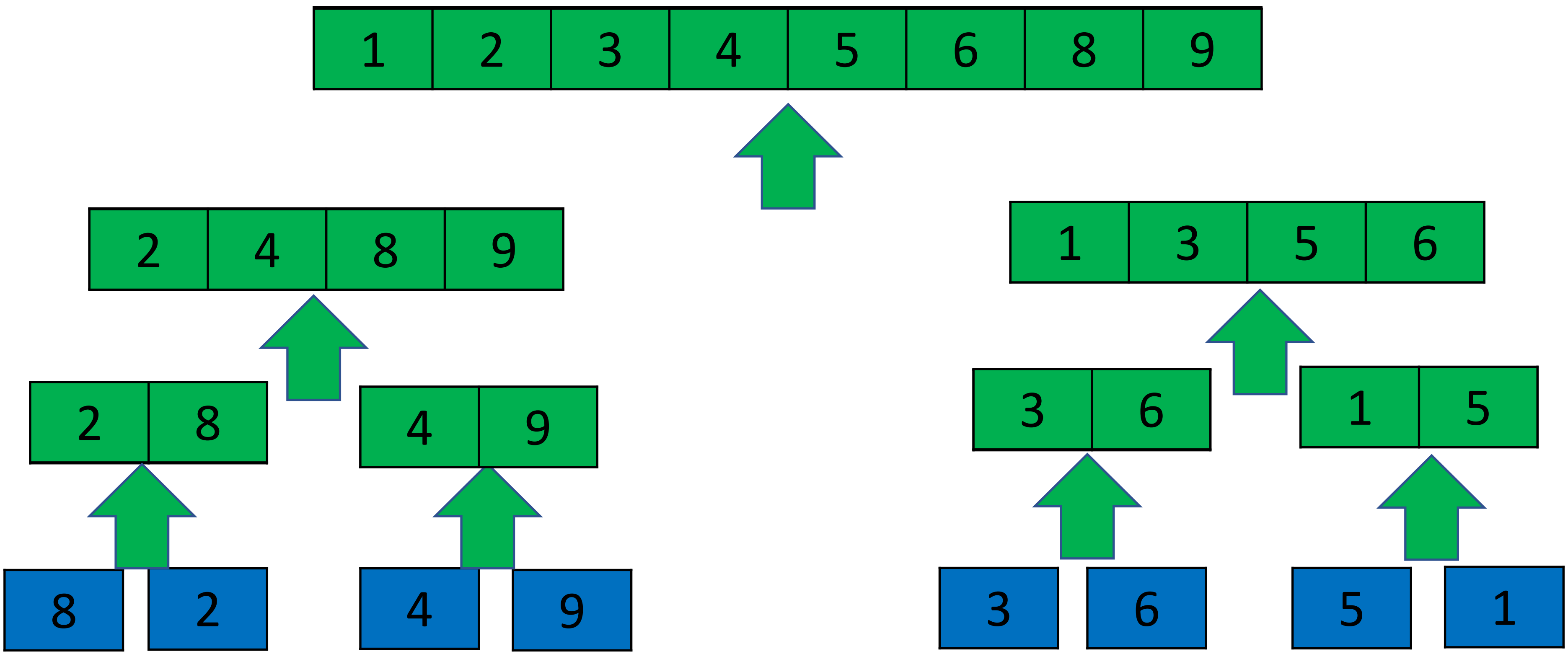
Mergesort



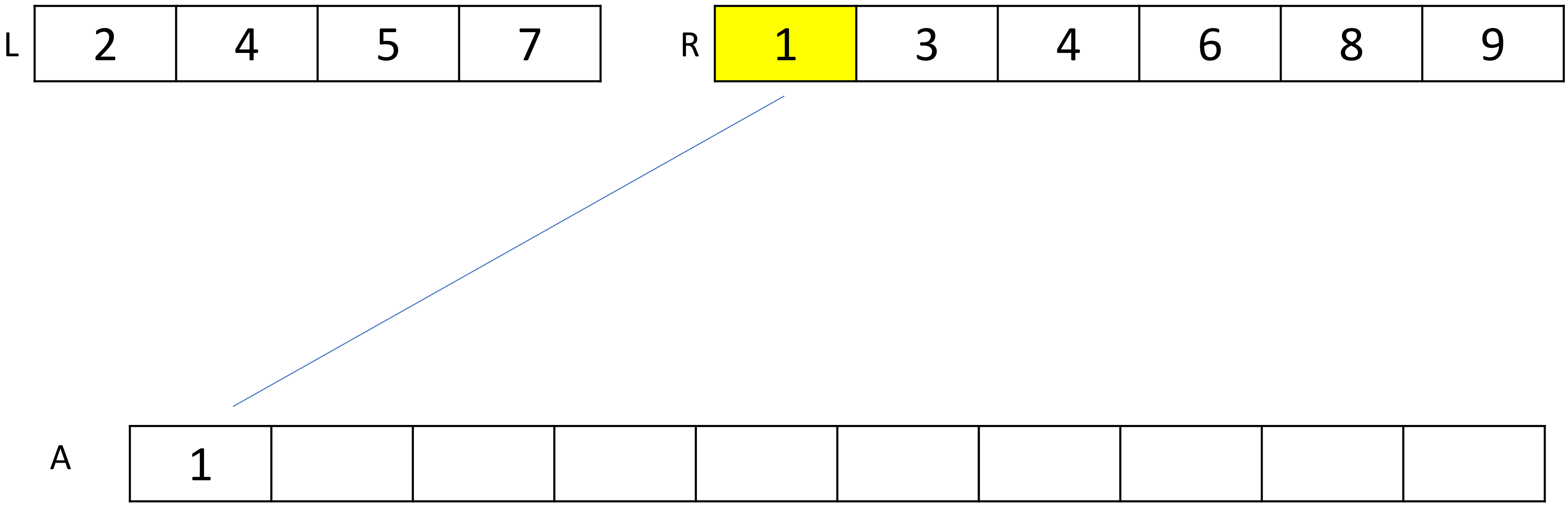
Mergesort



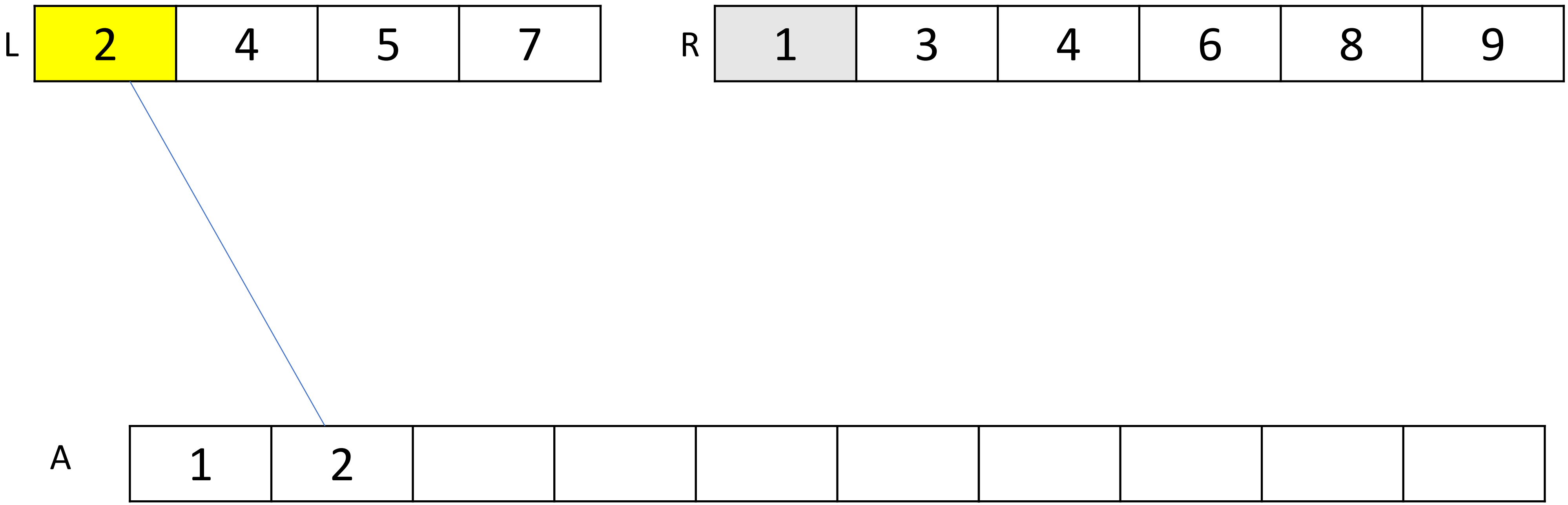
Mergesort



Merge two sorted arrays…



Merge two sorted arrays…



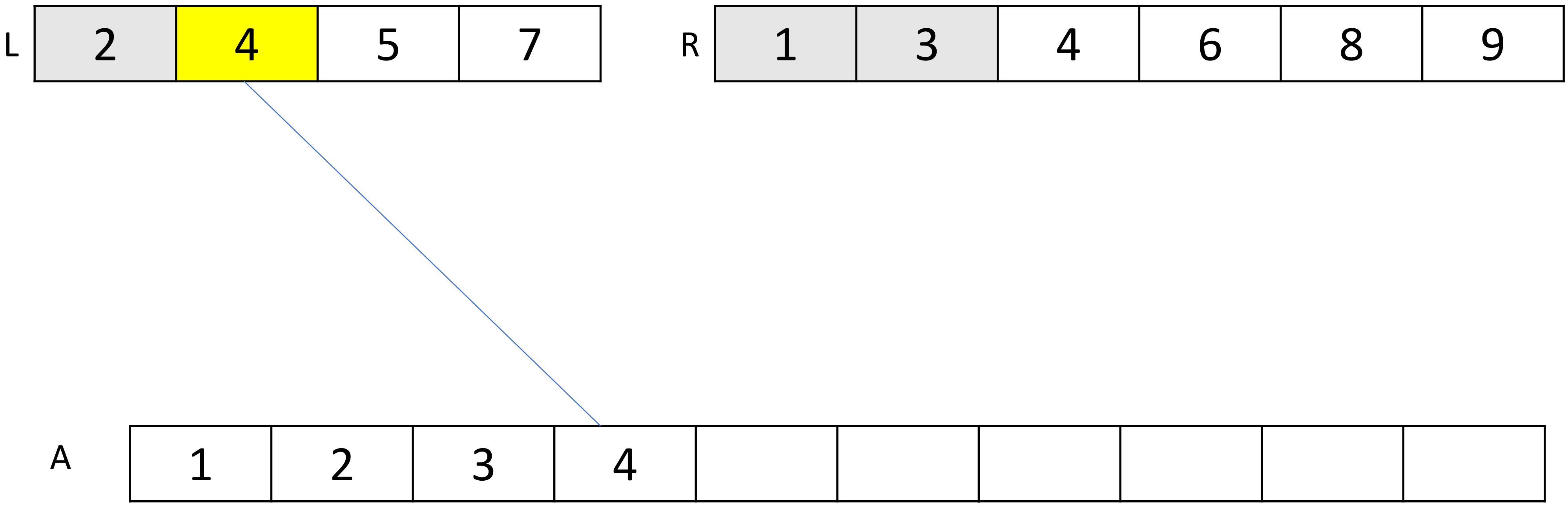
Merge two sorted arrays…

L	2	4	5	7

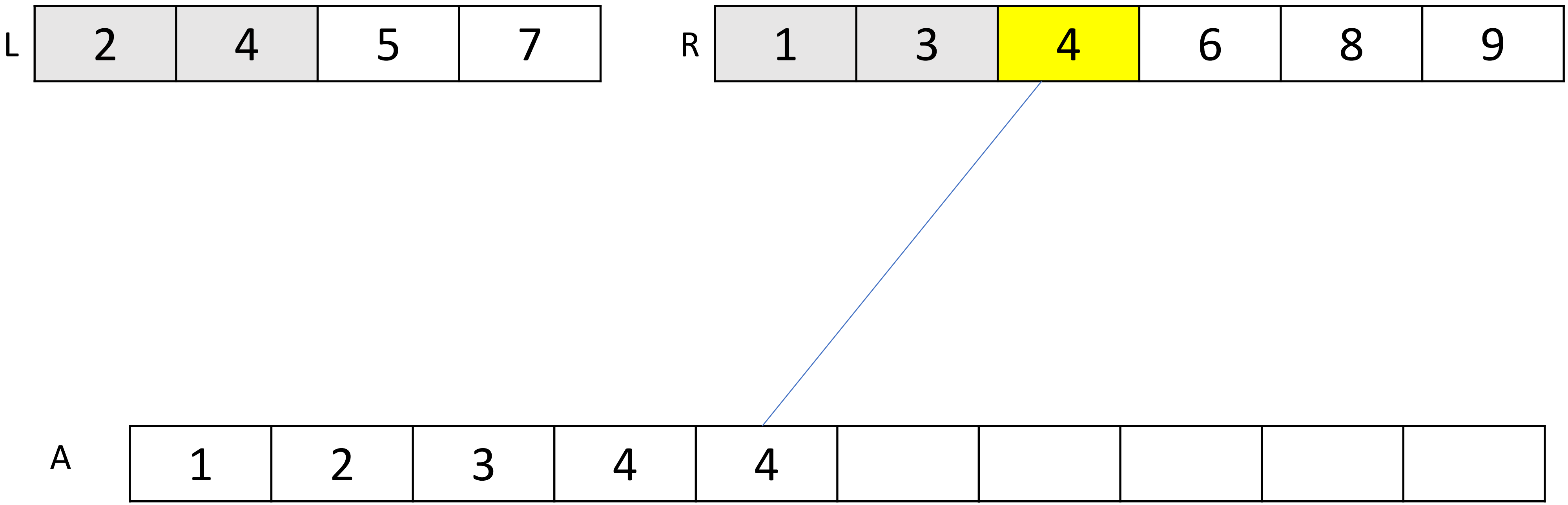
R	1	3	4	6	8	9

A	1	2	3							

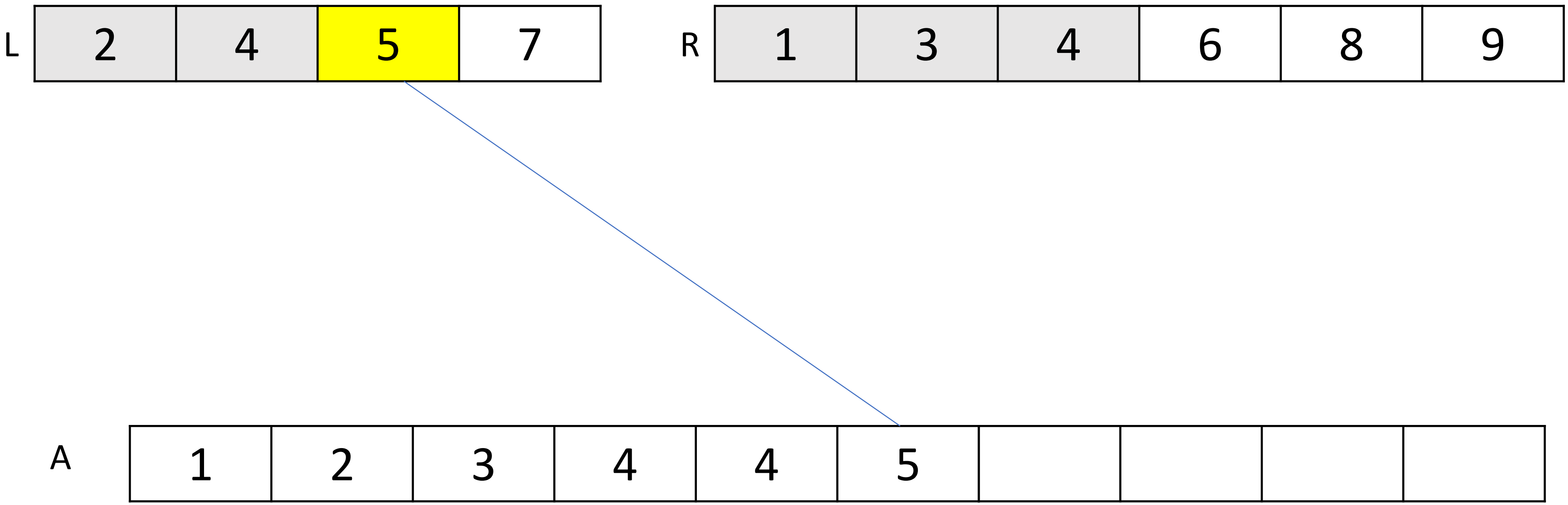
Merge two sorted arrays…



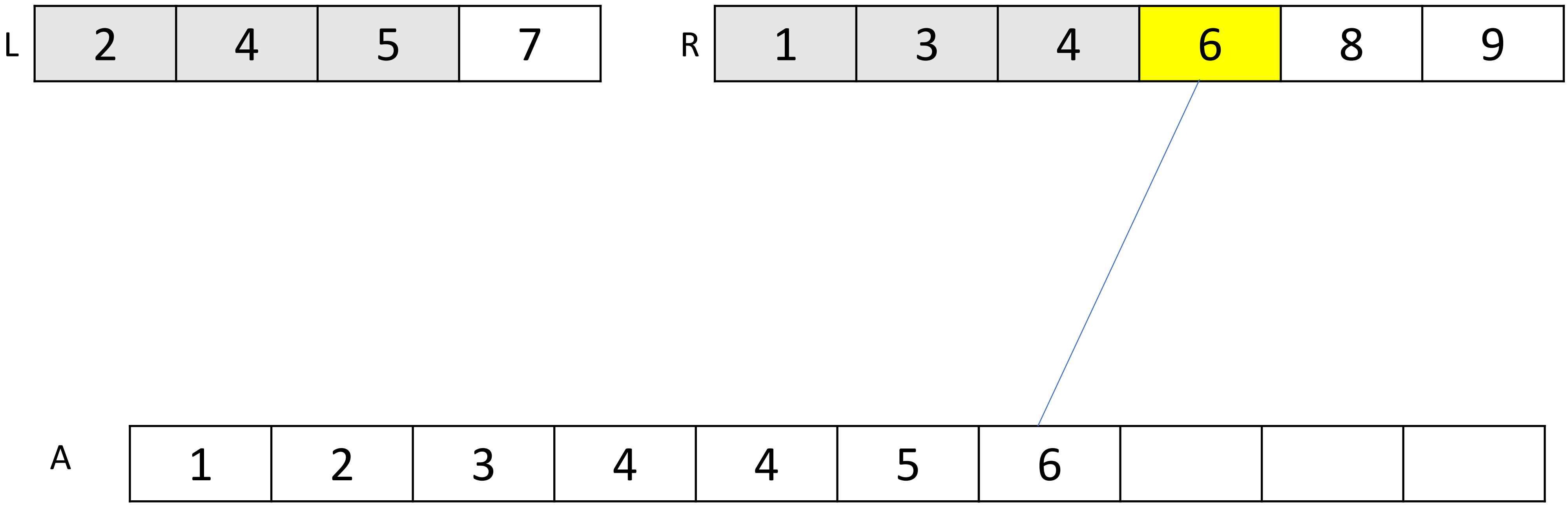
Merge two sorted arrays…



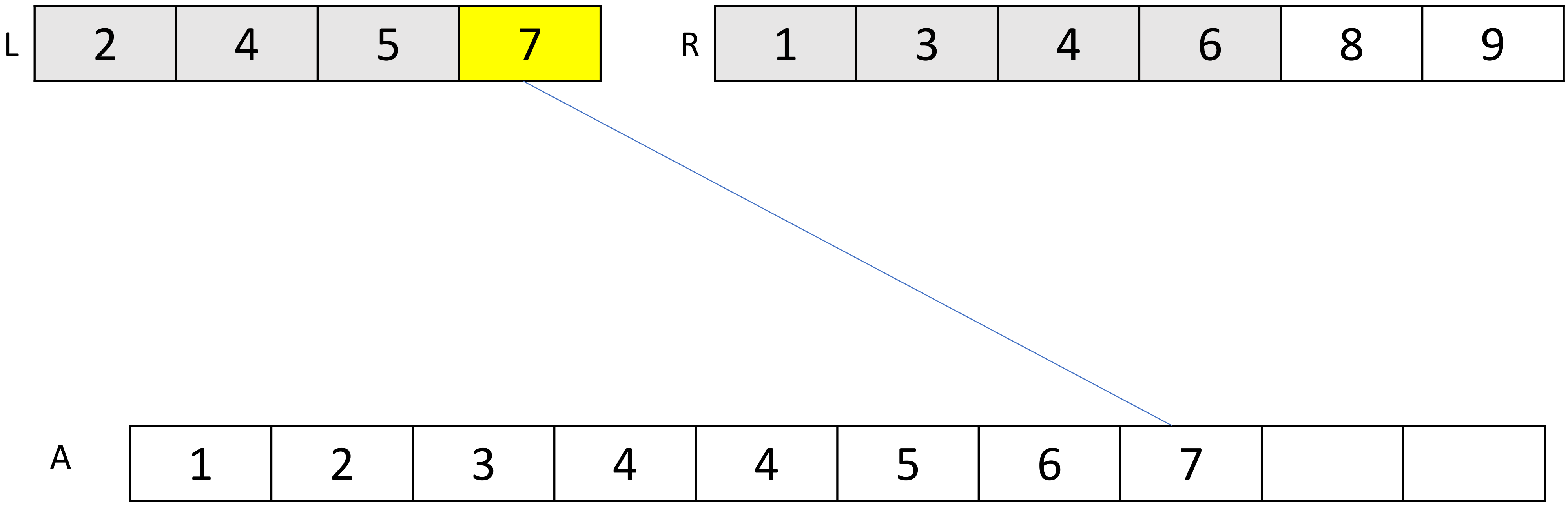
Merge two sorted arrays…



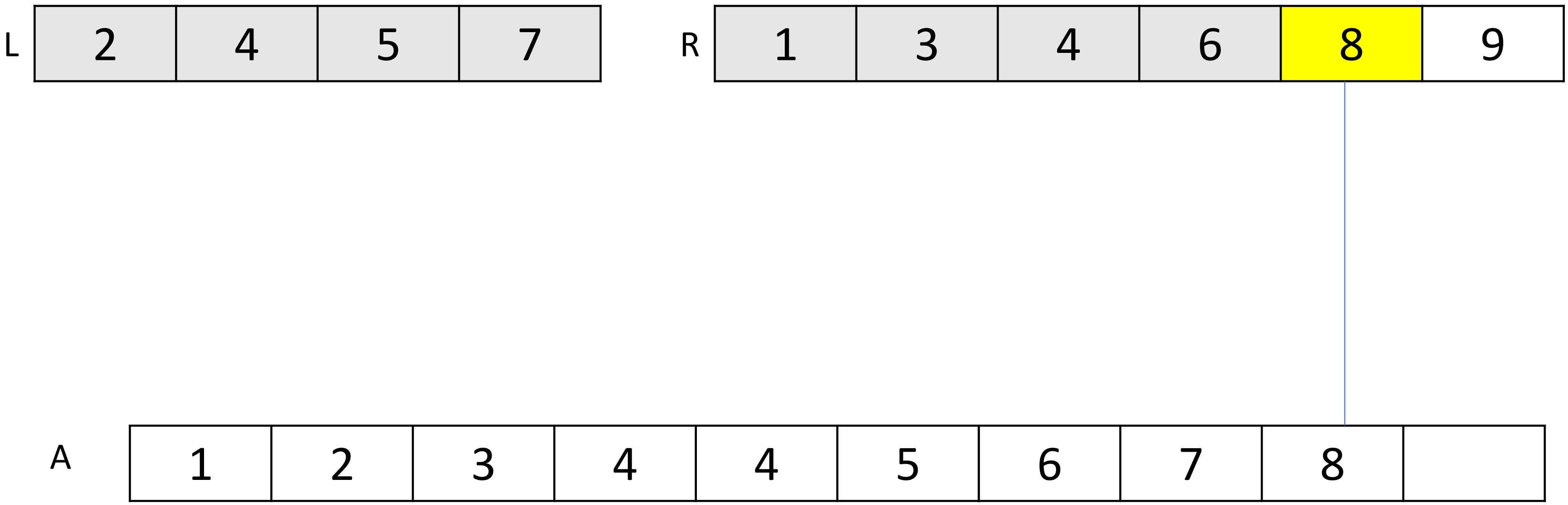
Merge two sorted arrays…



Merge two sorted arrays…



Merge two sorted arrays...



Merge two sorted arrays...

L	2	4	5	7

R	1	3	4	6	8	9

A	1	2	3	4	4	5	6	7	8	9

Merge A[p..q] with A[q+1..r] into A[p..r]

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i - 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else A[k] = R[j]
                j = j + 1
```

$\boldsymbol{Merge}$ $takes\ \theta(N)\ to\ merge\ N\ elements$

Mergesort

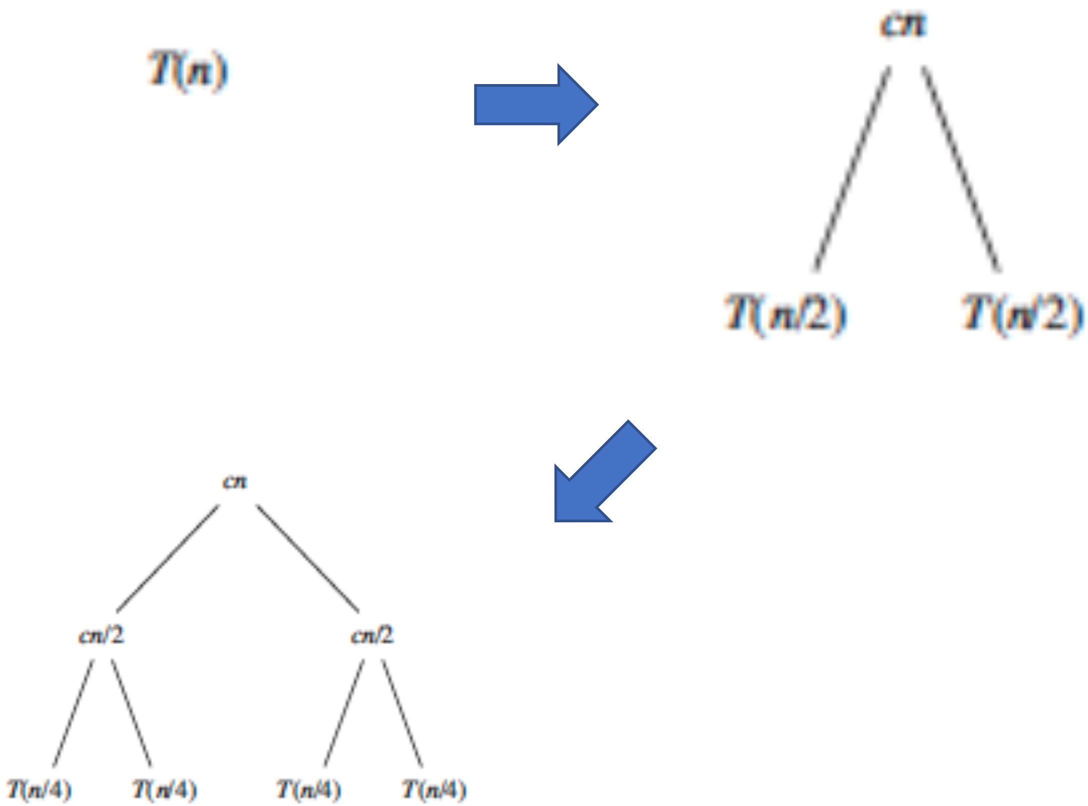
```
MERGE-SORT(A, p, r)

if p < r
    q = ⌊(p + r)/2⌋                θ(1)
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)        ~ 2T(n/2)
    MERGE(A, p, q, r)              θ(N)
```

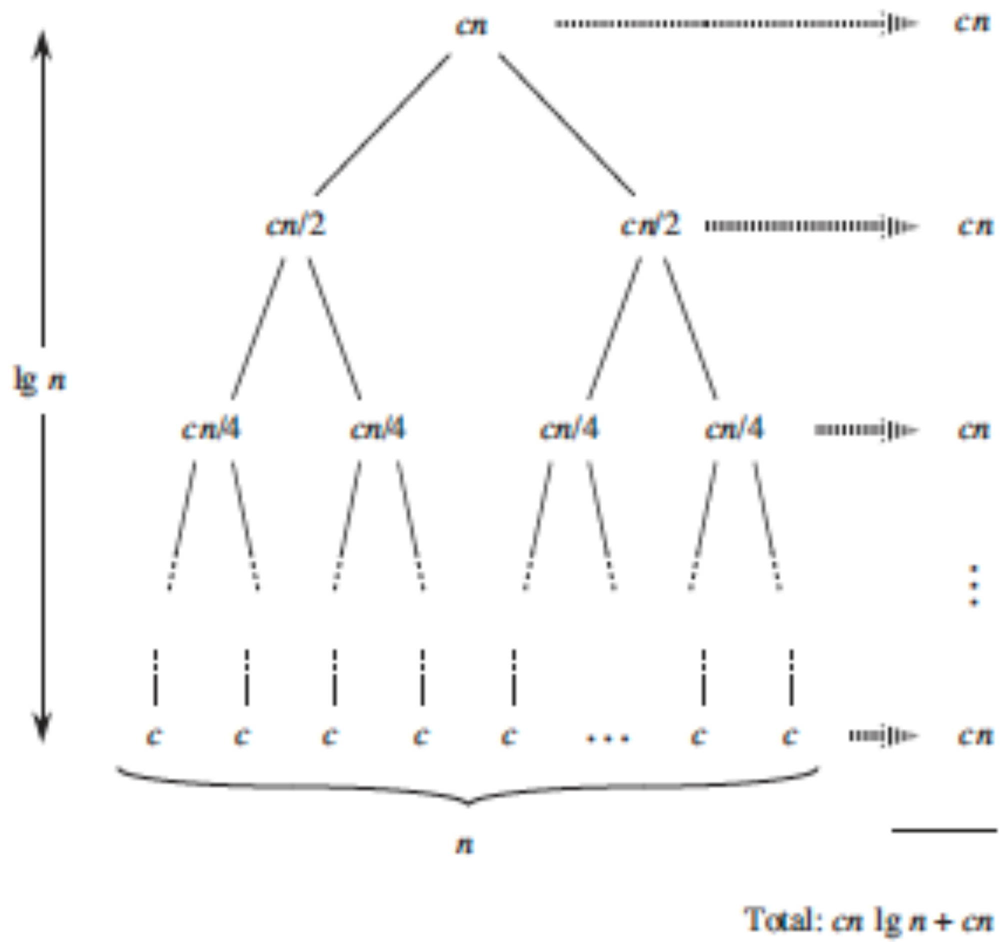
Running time of Mergesort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

Using Recursion Tree to solve for T(n)



Using Recursion Tree to solve for T(n)



* Mergesort running time is **linearithmic**. $T(N) = \theta(N\ lg\ N)$

Pop-Quiz

- Illustrate the operation of merge-sort on the array

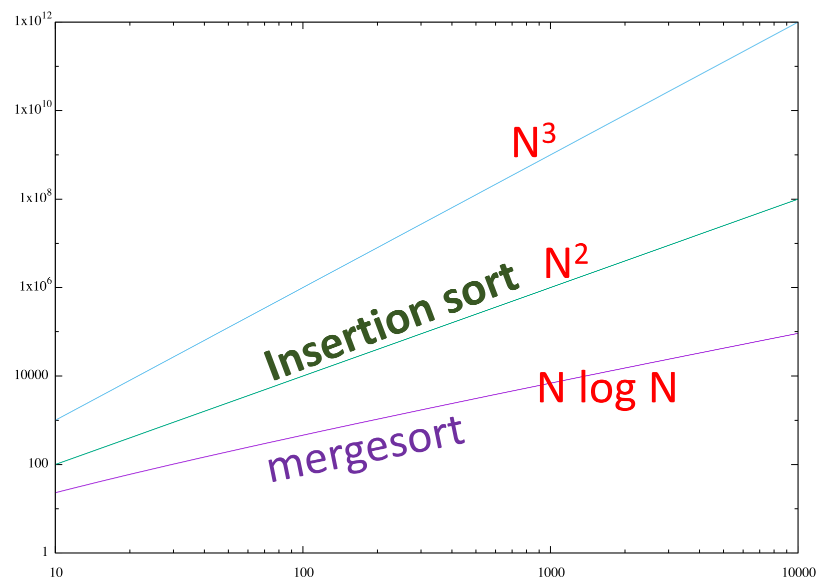
  A = <3, 41, 52, 26, 38, 57, 9, 49>

- Compare running time for different size of n:

n	$T(n) = 2n^2$	$T(n) = 8n \lg n$
2		
4		
8		
16		
32		
64		
128		

Summary: MergeSort

- Worst case running time is $\theta(\, N \lg N)$
- For large enough input size, mergesort always beat insertion sort.
- **<u>NOT</u>** sort "in place"
- Mergesort is not optimal with respect to space usage



Quicksort

- Proposed in 1961 by C.A.R. Hoare
- Divide-and-Conquer algorithm
- Sorts "in place" (like insertion sort)
- Highly practical (with appropriate tuning)

Tony Hoare
1980 Turing Award

ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure   quicksort (A,M,N);   value M,N;
                array A;   integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin       integer I,J;
            if M < N then begin partition (A,M,N,I,J);
                                quicksort (A,M,J);
                                quicksort (A, I, N)
                          end
end         quicksort
```

Communications of the ACM (July 1961)

Quicksort

- **Divide**: Partition the array into two subarrays around a *pivot* x such that elements in lower subarray ≤ x ≤ elements in upper subarray

- **Conquer**: Sort the two subsequences recursively using quicksort.
- **Combine**: -

Quicksort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

quicksort(A, 1, 8)

partition(A, p=1, r=8)

$i=0$ $j=1$ $x = A[8]$

8	2	1	9	3	6	5	4

$i=1$ $j=2$ $x = A[8]$

2	8	1	9	3	6	5	4

$i=2$ $j=3$ $x = A[8]$

2	1	8	9	3	6	5	4

$i=2$ $j=4$ $x = A[8]$

2	1	8	9	3	6	5	4

$i=3$ $j=5$ $x = A[8]$

2	1	3	9	8	6	5	4

quicksort(A, 1, 8)

i=3 *j=6* *x = A[8]*

2	1	3	9	8	6	5	4

partition(A, p=1, r=8)

i=3 *j=7* *x = A[8]*

2	1	3	9	8	6	5	4

i=3 *j=7* *x = A[8]*

2	1	3	4	8	6	5	9

return 4

quicksort(A, p=1, r=3) ***quicksort(A, p=5, r=8)***

quicksort(A, 1, 3)

i=0 *j=1* *x = A[3]*

2	1	3	4	8	6	5	9

partition(A, p=1, r=3)

i=1 *j=1* *x = A[3]*

2	1	3	4	8	6	5	9

i=2 *j=2* *x = A[3]*

2	1	3	4	8	6	5	9

return 3

quicksort(A, 1, 3)

quicksort(A, 1, 8)

quicksort(A, 1, 2)

i=0 *j=1* *x = A[2]*

2	1	3	4	8	6	5	9

partition(A, p=1, r=2)

j=2
i=0 *x = A[2]*

2	1	3	4	8	6	5	9

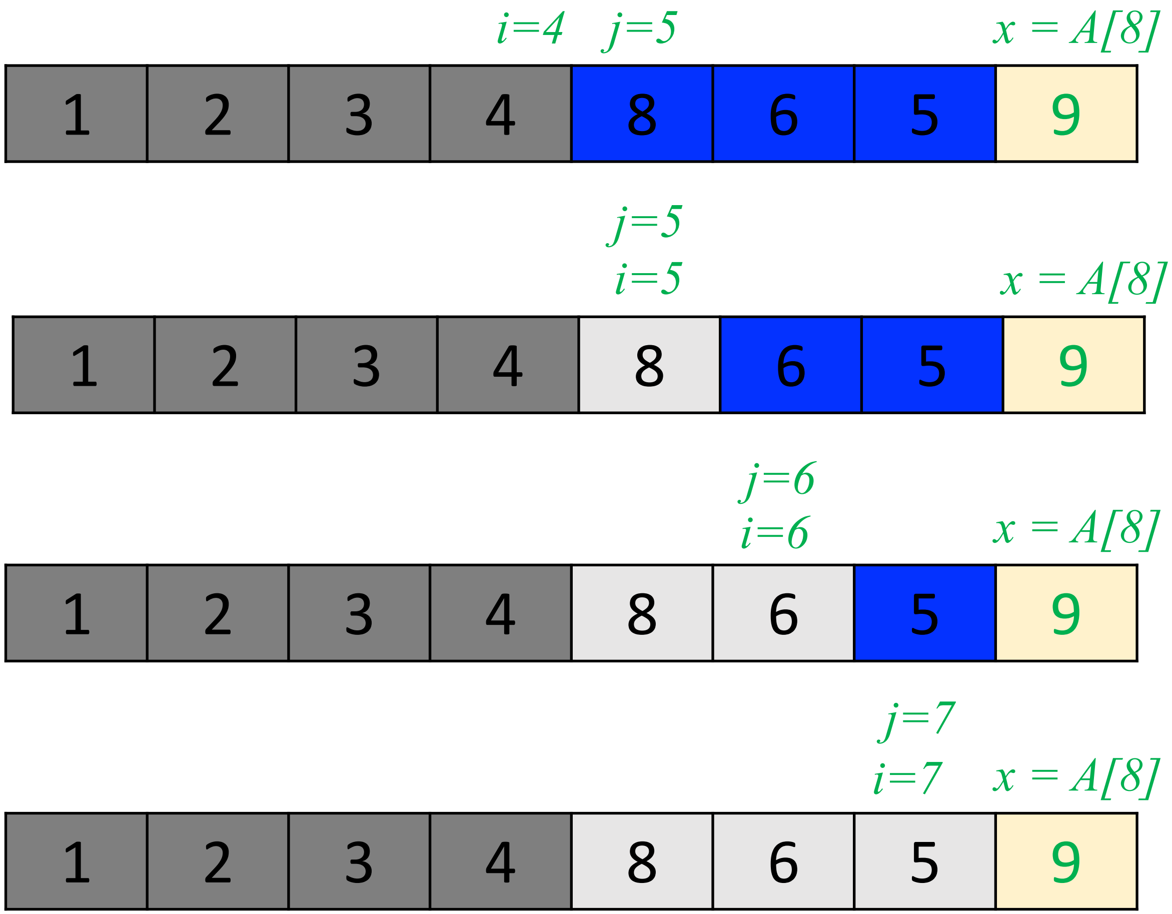
j=2
i=0 *x = A[2]*

1	2	3	4	8	6	5	9

return 1

quicksort(A, 1, 2)
quicksort(A, 1, 3)
quicksort(A, 1, 8)

quicksort(A, 5, 8)



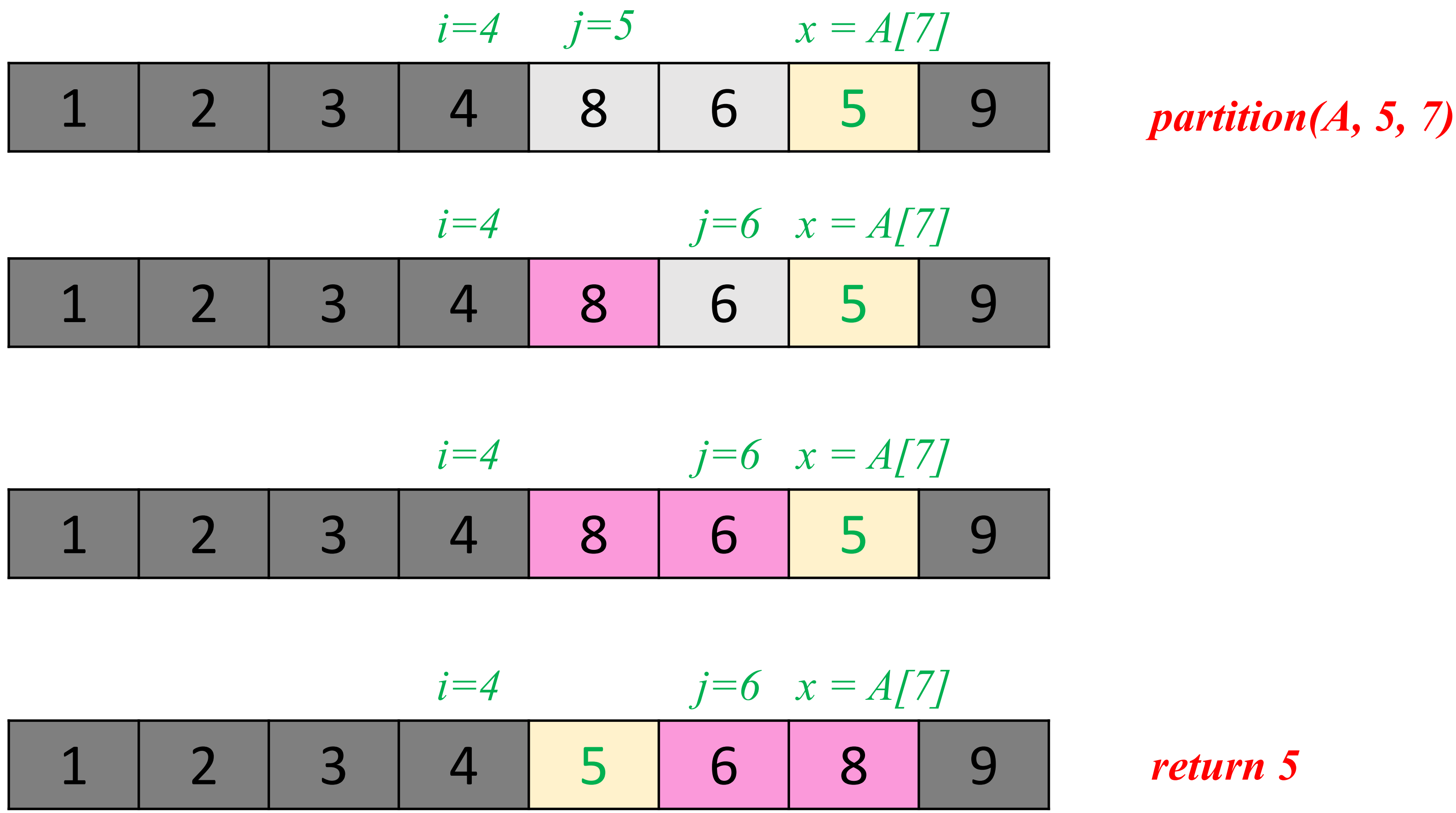
partition(A, p=5, r=8)

return 8

quicksort(A, 5, 8)

quicksort(A, 1, 8)

quicksort(A, 5, 7)



quicksort(A, 5, 7)

quicksort(A, 5, 8)

quicksort(A, 1, 8)

quicksort(A, 6, 7)

i=5 *j=6* *x = A[7]*

1	2	3	4	5	6	8	9

partition(A, 6,7)

i=6
j=6 *x = A[7]*

1	2	3	4	5	6	8	9

return 7

quicksort(A, 6, 7)

quicksort(A, 5, 7)

quicksort(A, 5, 8)

quicksort(A, 1, 8)

Sorted…

1	2	3	4	5	6	8	9

Worst Case Running Time of Quicksort

- Input already sorted or reverse sorted
- Partition around min/max

$$T(n) = T(0) + T(n\text{-}1) + \theta(\ N\ )$$

$$= \theta(1) + T(n-1) + \theta(\ N)$$

$$= T(n\text{-}1) + \theta(N)$$

$$= \theta(N^2)$$

Best Case Running Time of Quicksort

- Partition splits the array evenly

$$T(n) = 2T(n/2) + \theta(\ N\ )$$

$$= \theta(N \lg N)$$ .... same as mergesort

How to make sure that Partition always splits the array evenly ?

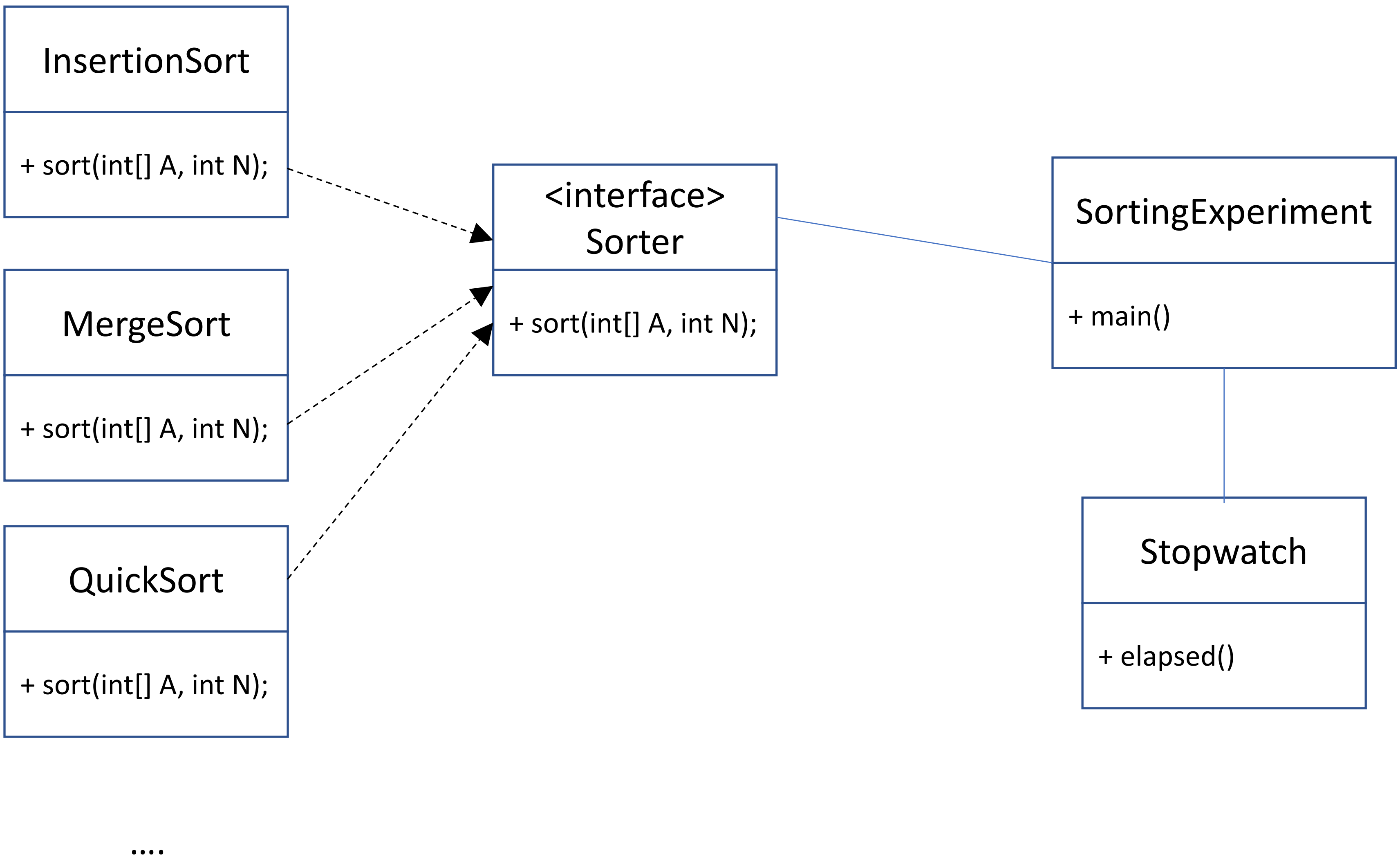
Randomized Quicksort

- **Idea**: Partition the array around a *random* element
- Running time is independent of the order of input
- The worst-case is determined only by the output of a random number generator

Summary: Quicksort

- A great general purpose sorting algorithm
- Sort “in place”
- Typically over twice as fast as mergesort
- Performance can be substantially improved by various code tuning

Java Implementation



```java
public Stopwatch() {
        start = System.nanoTime();
}
public long elapsed() {
        return (System.nanoTime() - start);
}
```

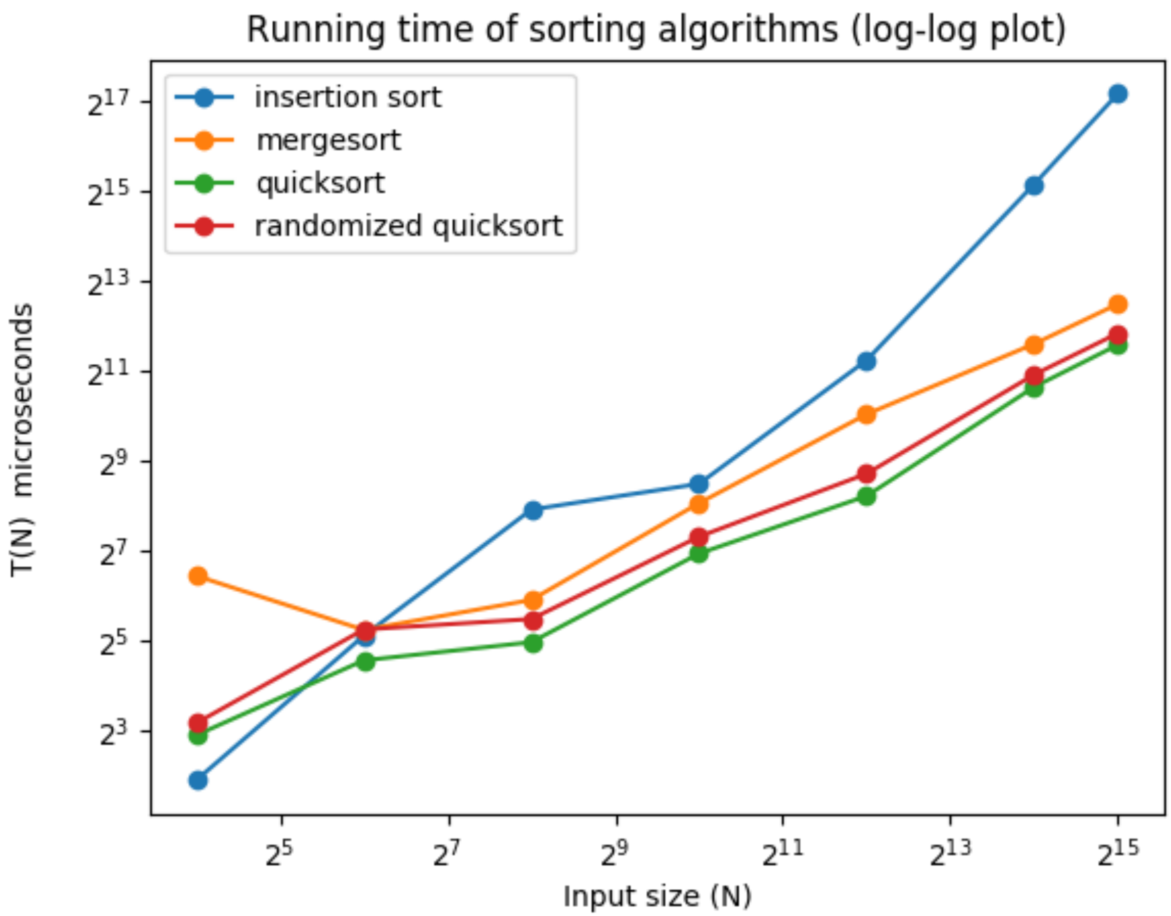
```java
/****** Measure Sorting Time ******/
w = new Stopwatch();
sorter.sort(A, N);
totTime += w.elapsed();
/**********************************/
```

```
kulwadee-mbair:int320 ann$ javac dsa/sorting/*.java
kulwadee-mbair:int320 ann$ java -cp . dsa.sorting.SortingExperiment
#Size: 16 64 256 1024 4096 16384 32768
#Time unit: microseconds
      insertion sort:       8.21      69.15     363.09     460.71    2513.76   36552.40  146835.95
           mergesort:     209.65      50.06      91.96     264.72    1105.28    3217.84    5638.38
           quicksort:       7.82      19.89      27.29     116.54     325.38    1432.15    3152.02
randomized quicksort:      10.23      40.85      44.50     143.15     588.93    1773.45    3640.32
```

```
java -cp . dsa.sorting.SortingExperiment > sorting.txt
```

```
python plot_sorting.py sorting.txt
```

output a log-log plot of the running time to `sorting.png`



Pop-Quiz

1. Given specifications of two computers *A* and *B*:

Computer A: can execute *2 billion instructions per second.*

Computer B: can execute *10 million instructions per second.*

Answer the following questions.

(a) Computer *A* is ______________________ times (*faster / slower*) than Computer *B*.

(b) Suppose that a program for Computer *A* that implements an *insertion sort* algorithm requires ***$4n^2$* instructions to sort *n* numbers**. How long does it take for Computer *A* to sort **10 million numbers using this insertion sort** program?

(c) Suppose that a program for Computer *B* that implements a *merge sort* algorithm requires ***$100n \log n$* instructions to sort *n* numbers**. How long does it take for Computer *B* to sort **10 million numbers using this merge sort** program?